John von Neumann Institute for Computing

Parallelization of the C++ Navier-Stokes Solver DROPS with OpenMP

C. Terboven, A. Spiegel, D. an Mey, S. Gross, V. Reichelt

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 431-438, 2006.

http://www.fz-juelich.de/nic-series/volume33

Parallelization of the C++ Navier-Stokes Solver DROPS with OpenMP

Christian Terboven[a], Alexander Spiegel[a], Dieter an Mey[a], Sven Gross[b], Volker Reichelt[b]

[a]Center for Computing and Communication, RWTH Aachen University, 52074 Aachen, Germany

[b]Institut für Geometrie und Praktische Mathematik, RWTH Aachen University, 52056 Aachen, Germany

1. Introduction

The Navier-Stokes solver DROPS [1] is developed at the IGPM (Institut für Geometrie und Praktische Mathematik) at the RWTH Aachen University, as part of an interdisciplinary project (SFB 540: Model-based Experimental Analysis of Kinetic Phenomena in Fluid Multi-phase Reactive Systems [2]) where complicated flow phenomena are investigated. The object-oriented programming paradigm offers a high flexibility and elegance of the program code, facilitating development and investigation of numerical algorithms. Template programming techniques and the C++ Standard Template Library (STL) are heavily used.

In cooperation with the Center for Computing and Communication of the RWTH Aachen University a detailed runtime analysis of the code has been carried out and the computationally dominant program parts have been tuned and parallelized with OpenMP.

The UltraSPARC IV- and Opteron-based Sun Fire SMP-Clusters have been the prime target platforms, but other architectures have been investigated, too. It turned out that the sophisticated usage of template programming in combination with OpenMP is quite demanding for many C++ compilers. We observed a high variation in performance and many compiler failures.

In chapter 2 the DROPS package is described briefly. In chapter 3 we take a look at the performance of the original and the tuned serial code versions. In chapter 4 we describe the OpenMP parallelization and its performance. Chapter 5 contains a summary of our findings.

2. The DROPS multi-phase Navier-Stokes solver

The aim of the ongoing development of the DROPS software package is to build an efficient software tool for the numerical simulation of three-dimensional incompressible multi-phase flows. More specifically, we want to support the modeling of complex physical phenomena like the behavior of the phase interface of liquid drops, mass transfer between drops and a surrounding fluid, or the coupling of fluid dynamics with heat transport in a laminar falling film by numerical simulation. Although quite a few packages in the field of CFD already exist, a black-box solver for such complicated flow problems is not yet available.

From the scientific computing point of view it is of interest to develop a code that combines the efficiency and robustness of modern numerical techniques, such as adaptive grids and iterative solvers, with the flexibility required for the modeling of complex physical phenomena.

For the simulation of two-phase flows we implemented a levelset technique for capturing the phase interface. The advantage of this method is that it mainly adds a scalar PDE to the Navier-Stokes system and therefore fits nicely into the CFD framework. But still, the coupling of the phase interface with the Navier-Stokes equations adds one layer of complexity.

Several layers of nesting in the solvers induced by the structure of the mathematical models require fast inner-most solvers as well as fast discretization methods since many linear systems have

to be regenerated in each time step. Apart from the numerical building blocks, software engineering aspects such as the choice of suitable data structures, in order to decouple the grid generation and finite element discretization (using a grid based data handling) as much as possible from the iterative solution methods (which use a sparse matrix format), are of main importance for performance reasons.

3. Portability and Performance of the Serial Program Version

3.1. Platforms

The main development platform of the IGPM is a standard PC running Linux using the popular GNU C++ compiler [3]. Because this compiler does not support OpenMP, we had to look for adequate C++ compilers supporting OpenMP on our target platforms. Table 1 lists compilers and machines which we considered for our tuning and parallelization efforts. It also introduces abbreviations for each hardware-compiler-combination, which will be referred to as ”platforms” in the remainder of the paper.

The programming techniques employed in the DROPS package (Templates, STL) caused quite some portability problems due to lacking standard conformance of the compilers, therefore the code had to be patched for most compilers. Unfortunately not all of the available OpenMP-aware compilers were able to successfully compile the final OpenMP code version.

From the early experiences gathered by benchmarking the original serial program and because of the good availability of the corresponding hardware, we concentrated on the OPT+icc and USIV+guide platforms for the development of the OpenMP version and recently on OPT+ss10 and USIV+ss10.

3.2. Runtime profile

The runtime analysis (USIV+guide) shows that assembling the stiffness matrices (SETUP) costs about 52% of the total runtime, whereas the PCG-method including the sparse-matrix-vector-multiplication costs about 21% and the GMRES-method about 23%. Together with the utility routine LINCOMB these parts of the code account for 99% of the total runtime. All these parts have been considered for tuning and for parallelization with OpenMP.

It must be pointed out that the runtime profile heavily depends on the number of mesh refinements and on the current timesteps. In the beginning of a program run the PCG-algorithm and the matrix-vector-multiplication take about 65% of the runtime, but because the number of iterations for the solution of the linear equation systems shrinks over time, the assembly of the stiffness matrices is getting more and more dominant. Therefore we restarted the program after 100 time steps and let it run for 10 time steps with 2 grid refinements for our comparisons.

3.3. Data Structures

In the DROPS package the Finite Element Method is implemented. This includes repeatedly setting up the stiffness matrices and then solving linear equation systems with PCG- and GMRES-methods.

Since the matrices arising from the discretization are sparse, an appropriate matrix storage format, the CRS (compressed row storage) format is used, in which only nonzero entries are stored. It contains an array *val* - which will be referred to later - for the values of the nonzero entries and two auxiliary integer arrays that define the position of the entries within the matrix. The data structure is mainly a wrapper class around a `valarray<double>` object, a container of the C++ Standard Template Library (STL).

Unfortunately, the nice computational and storage properties of the CRS format are not for free. A

machine	platform	compiler	runtime[s] original	runtime[s] tuned	OpenMP support
Standard PC: 2x Intel Xeon 2.66 GHz Hyper-Threading	XEON+gcc333	GNU C++ V3.3.3[3]	3694.9	1844.3	no
	XEON+gcc343	GNU C++ V3.4.3	2283.3	1780.7	no
	XEON+icc81	Intel C++ V8.1[4]	2643.3	1722.9	yes
	XEON+pgi60	PGI C++ V6.0-1	8680.1	5080.2	yes
Sun Fire V40z: 4x AMD Opteron 2.2 GHz	OPT+gcc333	GNU C++ V3.3.3	2923.3	1580.3	no
	OPT+gcc333X	GNU C++ V3.3.3 64bit	2167.8	1519.5	no
	OPT+icc81	Intel C++ V8.1	2404.0	1767.1	yes
	OPT+icc81X	Intel C++ V8.1 64bit	2183.4	1394.0	fails
	OPT+pgi60	PGI C++ V6.0-1[5]	6741.7	5372.9	yes
	OPT+pgi60X	PGI C++ V6.0-1 64bit	4755.1	3688.4	yes
	OPT+path20	PathScale EKOpath 2.0[6]	2819.3	1673.1	fails
	OPT+path20X	PathScale EKOpath 2.0, 64bit	2634.5	1512.3	fails
	OPT+ss10	Sun Studio C++ V10 update1[7]	3613.1	2431.0	yes
Sun Fire E2900: 12x UltraSPARC IV 1.2 GHz (dual core)	USIV+gcc331	GNU C++ V3.3.1	9782.4	7845.4	no
	USIV+ss10	Sun Studio C++ V10 update1	7673.7	4958.7	yes
	USIV+guide	Intel-KSL Guidec++ V4.0[8]	7551.0	5335.0	yes
IBM p690: 16x Power4 1.7 GHz (dual core)	POW4+guide	Intel-KSL Guidec++ V4.0	5535.1	2790.3	yes
	POW4+gcc343	GNU C++ V3.4.3	3604.0	2157.8	no
SGI Altix 3700: 128x Itanium2 1.3 GHz	IT2+icc81	Intel C++ V8.1	9479.0	5182.8	fails

Table 1
Compilers and machines, runtime of original and tuned serial versions, OpenMP support.
Linux is running on all platforms, except for OPT+ss10, USIV+* (Solaris 10) and POW4+* (AIX).

disadvantage of this format is that insertion of a non-zero element into the matrix is rather expensive. Since this is unacceptable when building the matrix during the discretization step, a sparse matrix builder class has been designed with an intermediate storage format based on STL's `map` container that offers write access in logarithmic time for each element. After the assembly, the matrix is converted into the CRS format in the original program version.

3.4. Serial Tuning Measures

On the Opteron systems the PCG-algorithm including a sparse-matrix-vector-multiplication and the preconditioner profit from manual prefetching. The performance gain of the matrix-vector-

multiplication is 44% on average, and the speedup of the preconditioner is 19% on average, depending on the addressing mode (64bit mode profits slightly more than 32bit mode).

As the setup of the stiffness matrix turned out to be quite expensive, we reduced the usage of the `map` datatype. As long as the structure of the matrix does not change, we reuse the index vectors and only fill the matrix with new data values. This leads to a performance plus of 50% on the USIV+guide platform and about 58% on the OPT+icc platform. All other platforms benefit from this tuning measure as well.

Table 1 lists the results of performance measurements of the original serial version and the tuned serial version. Note that on the Opteron the 64bit addressing mode typically outperforms the 32bit mode, because in 64bit mode the Opteron offers more hardware registers and provides an ABI which allows for passing function parameters using these hardware registers. This outweights the fact that 64bit addresses take more cache space.

4. The OpenMP Approach

4.1. Assembly of the Stiffness Matrices

The routines for the assembly of the stiffness matrices typically contain loops like the following:

```
for (MultiGridCL::const_TriangTetraIteratorCL
   sit=_MG.GetTriangTetraBegin(lvl),
   send=_MG.GetTriangTetraEnd(lvl);
   sit != send; ++sit)
```

Such a loop construct cannot be parallelized with a `for`-worksharing construct in OpenMP, because the loop iteration variable is not of type integer. We considered three ways to parallelize these loops:

- The `for`-loop is placed in a parallel region and the loop-body is placed in a `single`-worksharing construct whose implicit barrier is omitted by specifying the `nowait`-directive. The problem with this approach is that the overhead at the entry of the `single`-region limits the possible speedup.

- Intel's compilers and the guidec++ compiler offer the task-queuing construct as an extension to the OpenMP standard. For each value of the loop iteration variable the loop-body is enqueued in a work-queue by one thread and then dequeued and processed by all threads. The number of loop iterations is rather high in relation to the work in the loop body, so again the administrative overhead limits the speedup. We proposed an extension of the task-queuing construct implemented by Intel for the upcoming OpenMP standard version 3 for which a schedule clause with a chunksize can be specified.

- The pointers of the iterators are stored in an array in an additional loop, so that afterwards a simpler loop running over the elements of this array can be parallelized with a `for`-worksharing construct. We found this approach to be the most efficient giving the highest speedup.

Reducing the usage of the `map` STL datatype during the stiffness matrix setup as described in chapter 3 turned out to cause additional complexity and memory requirements in the parallel version. In the parallel version each thread fills a private temporary container consisting of one map per matrix row. The structure of the complete stiffness matrix has to be determined, which can be parallelized over the matrix rows. The master thread then allocates the `valarray` STL objects. Finally, the matrix rows are summed up in parallel.

If the structure of the stiffness matrix does not change, each thread fills a private temporary container consisting of one `valarray` of the same size as the array *val* of the final matrix.

This causes massive scalability problems for the guidec++-compiler. Its STL library obviously uses critical regions to be threadsafe. Furthermore the guidec++ employs an additional allocator for small objects which adds more overhead because of internal synchronization. Therefore we implemented a special allocator and linked to the Sun-specific memory allocation library *mtmalloc* which is tuned for multithreaded applications to overcome this problem.

Thus, the matrix assembly could be completely parallelized, but the scalability is limited, because the overhead increases with the number of threads used (see table 2). The parallel algorithm executed with only one thread performs worse than the tuned serial version on most platforms, because the parallel algorithm contains the additional summation step as described above. On the USIV+guide platform it scales well up to about eight threads, but then the overhead which is caused by a growing number of dynamic memory allocations and memory copy operations increases. Therefore we limited the number of threads used for the SETUP routines to a maximum of eight in order to prevent a performance decrease for a higher thread count. On the USIV+ss10 and POW4+guide platforms there is still some speedup with more threads. Table 2 shows the runtime of the matrix setup routines. Note, that on the XEON-icc81 platform Hyper-Threading is profitable for the matrix setup.

code	**serial**	**serial**	**parallel**				
	original	**tuned**	**1**	**2**	**4**	**8**	**16**
XEON+icc81	1592	816	1106	733	577	n.a.	n.a.
OPT+icc81	1363	793	886	486	282	n.a.	n.a.
OPT+ss10	2759	1154	1233	714	428	n.a.	n.a.
USIV+guide	4512	2246	2389	1308	745	450	460
USIV+ss10	4564	1924	2048	1435	796	460	314
POW4+guide	4983	2236	2176	1326	774	390	185

Table 2
C++ + OpenMP: matrix setup, runtime [s]

4.2. The Linear Equation Solvers

In order to parallelize the PCG- and GMRES-methods, matrix and vector operations, which beforehand had been implemented using operator overloading, had to be rewritten with C-style `for` loops directly accessing the structure elements. Thereby some synchronizations could be avoided and some parallelized `for`-loops could be merged.

The parallelized linear equation solvers including the sparse-matrix-vector-multiplication scale quite well, except for the intrinsic sequential structure of the Gauss-Seidel preconditioner which can only be partially parallelized. Rearranging the operations in a blocking scheme improves the scalability (`omp_block`) but still introduces additional organization and synchronization overhead.

A modified parallelizable preconditioner (`jac0`) was implemented which affects the numerical behavior. It leads to an increase in iterations to fulfill the convergence criterium. Nevertheless it leads to an overall improvement with four or more threads.

The straight-forward parallelization of the sparse matrix vector multiplication turned out to have a load imbalance. Obviously the nonzero elements are not equally distributed over the rows. The load balancing could be easily improved by setting the loop scheduling to `SCHEDULE(STATIC,128)`.

The linear equation solvers put quite some pressure on the memory system. This clearly reveals the memory bandwidth bottleneck of the dual processor Intel-based machines (XEON+icc). The ccNUMA-architecture of the Opteron-based machines (OPT+icc) exhibits a high memory bandwidth if the data is properly allocated. But it turns out that the OpenMP version of DROPS suffers from the fact that most of the data is allocated in the master thread's memory because of the usage of the STL datatypes.

As an experiment we implemented a C++ version of the stream benchmark using the STL datatype `valarray` on one hand and simple C-style arrays on the other hand. These arrays are allocated with *malloc* and initialized in a parallel region. Table 3 lists the memory bandwidth in GB/s for one of the kernel loops (saxpying) and a varying number of threads. It is obvious that the memory bandwidth does not scale when `valarrays` are used. The master thread allocates and initializes (after construction a `valarray` has to be filled with zeros by default) a contiguous memory range for the `valarray` and because of the first touch memory allocation policy, all memory pages are put close to the master thread's processor. Later on, all other threads have to access the master thread's memory in parallel regions thus causing a severe bottleneck.

The Linux operating system currently does not allow an explicit or automatic data migration. The Solaris operating system offers the Memory Placement Optimization feature (MPO), which can be used for an explicit data migration. In our experiment we measured the kernels using `valarrays` after the data has been migrated by a "next-touch" mechanism using the *madvise* runtime function, which clearly improves parallel performance (see table 3). This little test demonstrates how sensitive the Opteron architecture reacts to disadvantageous memory allocation and how a "next-touch" mechanism can be employed beneficially. We proposed a corresponding enhancement of the OpenMP specification in the upcoming Version 3.0. On the USIV+guide, USIV+ss10 and OPT+ss10 platforms we were able to exploit the MPO feature of Solaris to improve the performance of DROPS.

Stream kernel	**Data structure**	**Initialization method**	**1 Thread**	**2 Threads**	**3 Threads**	**4 Threads**
saxpying	valarray	implicit	2.11	2.16	2.15	2.03
	valarray	implicit+madvise	2.10	4.18	6.20	8.20
	C-array	explicit parallel	2.15	4.26	6.30	8.34

Table 3
Stream benchmark, C++ (valarray) vs. C, memory bandwidth in GB/s on OPT+ss10

On the whole the linear equation solvers scale reasonably well, given that frequent synchronizations in the CG-type linear equation solvers are inevitable. The modified preconditioner takes more time than the original recursive algorithm for few threads, but it pays off for at least four threads. Table 4 shows the runtime of the solvers.

4.3. Total Performance

Table 5 shows the total runtime of the DROPS code on all platforms for which a parallel OpenMP version could be built. Please note that we didn't have exclusive access to the POW4 platform. Table 6 shows the resulting total speedup.

code	serial	serial	parallel (omp_block)				parallel (jac0)			
	original	tuned	1	2	4	8	1	2	4	8
XEON+icc81	939	894	746	593	780	n.a.	837	750	975	n.a.
OPT+icc81	901	835	783	541	465	n.a.	668	490	415	n.a.
OPT+ss10	731	673	n.a.	n.a.	n.a.	n.a.	596	375	278	n.a.
USIV+guide	2682	2727	2702	1553	1091	957	1563	902	524	320
USIV+ss10	2714	2652	2870	1633	982	816	2555	1314	659	347
POW4+guide	440	441	589	315	234	159	572	331	183	113

Table 4
C++ + OpenMP: linear equation solvers, runtime [s]

code	serial	serial	parallel (omp_block)				parallel (jac0)				
	original	tuned	1	2	4	8	1	2	4	8	16
XEON+icc81	2643	1723	2001	1374	1353	n.a.	2022	1511	1539	n.a.	n.a.
OPT+icc81	2404	1767	1856	1233	952	n.a.	1738	1162	891	n.a.	n.a.
OPT+ss10	3613	2431	n.a.	n.a.	n.a.	n.a.	1973	1234	856	n.a.	n.a.
USIV+guide	7551	5335	5598	3374	2319	1890	4389	2659	1746	1229	1134
USIV+ss10	7674	4959	5422	3573	2255	1736	5056	3214	1894	1250	956
POW4+guide	5535	2790	3017	1752	1153	655	2885	1084	1099	641	482

Table 5
C++ + OpenMP: total runtime, runtime [s]

code	serial	serial	parallel (omp_block)				parallel (jac0)				
	original	tuned	1	2	4	8	1	2	4	8	16
XEON+icc81	1.00	1.53	1.32	1.92	1.95	n.a.	1.31	1.75	1.72	n.a.	n.a.
OPT+icc81	1.00	1.36	1.30	1.95	2.53	n.a.	1.38	2.07	2.70	n.a.	n.a.
OPT+ss10	1.00	1.49	n.a.	n.a.	n.a.	n.a.	1.83	2.93	4.22	n.a.	n.a.
USIV+guide	1.00	1.42	1.35	2.24	3.26	3.99	1.72	2.84	4.32	6.14	6.66
USIV+ss10	1.00	1.55	1.42	2.15	3.40	4.42	1.52	2.39	4.05	6.14	8.03
POW4+guide	1.00	1.98	1.83	3.16	4.80	9.73	1.92	3.07	5.04	8.63	11.48

Table 6
C++ + OpenMP: speed-up

5. Summary

The compute intense program parts of the DROPS Navier-Stokes solver have been tuned and parallelized with OpenMP. The heavy usage of templates in this C++ program package is a challenge for many compilers. As not all C++ compilers support OpenMP, and some of those which do, fail for the parallel version of DROPS, the number of suitable platforms turned out to be limited.

We ended up with using the guidec++ compiler from KAI (which is now part of Intel) and the Sun Studio 10 compilers on our UltraSPARC IV-based and Opteron-based Sun Fire servers and the Intel compiler in 32 bit mode on our Opteron-based Linux cluster.

The strategy which we used for the parallelization of the Finite Element Method implemented

in DROPS was straight forward. Nevertheless the obstacles which we encountered were manifold, many of them are not new to OpenMP programmers.

OpenMP programs running on big server machines operating in multi-user mode suffer from a high variation in runtime. Thus it is hard to see clear trends concerning speed-up. Exclusive access to the UltraSPARC IV-, Opteron- and Xeon-based systems helped a lot. On the 4-way Opteron systems the *taskset* Linux command was helpful to get rid of negative process scheduling effects.

We obtained the best absolute performance and the best parallelization speed-up on the POW4+guide platform, using a compiler which is no longer available. The best OpenMP version runs 11.5 times faster with 16 threads than the original serial version on the same platform (POW4+guide). But as we improved the serial version during the tuning and parallelization process, the speed-up compared to the tuned serial version is only 5.8.

On the OPT+ss10 platform, the best OpenMP version runs 4.2 faster than the original serial version and 2.8 faster than the tuned serial version with four threads.

An Opteron processor outperforms a single UltraSPARC IV processor core by about a factor of four. As Opteron processors are not available in large shared memory machines and scalability levels off with more than sixteen threads, shorter elapsed times are currently not attainable.

Acknowledgements
The authors would like to thank Uwe Mordhorst at University of Kiel and Bernd Mohr at Research Center Jülich for granting access to and supporting the usage of their machines, an SGI Altix 3700 and an IBM p690, respectively.

References

[1] Sven Gross, Jörg Peters, Volker Reichelt, Arnold Reusken: The DROPS package for numerical simulations of incompressible flows using parallel adaptive multigrid techniques. ftp://ftp.igpm.rwth-aachen.de/pub/reports/pdf/IGPM211_N.pdf
[2] Arnold Reusken, Volker Reichelt: Multigrid Methods for the Numerical Simulation of Reactive Multiphase Fluid Flow Models (DROPS). http://www.sfb540.rwth-aachen.de/Projects/tpb4.php
[3] GNU Compiler documentation: http://gcc.gnu.org/onlinedocs/
[4] Intel C/C++ Compiler documentation: http://support.intel.com/support/performancetools/c/linux/manual.htm
[5] PGI-Compiler documentation: http://www.pgroup.com/resources/docs.htm
[6] Pathscale-Compiler: http://www.pathscale.com
[7] Sun Studio 10: http://developers.sun.com/prodtech/cc/documentation/ss10_docs/
[8] Guide-Compiler of the KAP Pro/Toolset: http://developer.intel.com/software/products/kappro/